The Design of a 3D Rendering Meta System

Jürgen Döllner and Klaus Hinrichs
Institut für Informatik, Universität Münster
Einsteinstr. 62, D-48149 Münster
{dollner, khh}@uni-muenster.de

Abstract. This paper describes the design of a 3D rendering meta system which wraps the functionality of 3D rendering packages into a uniform, object-oriented framework. The homogeneous interface of the 3D rendering meta system serves as an easy-to-learn application programming interface to 3D rendering packages and allows us to exchange rendering packages without the need to recode an application. Our approach is based on a logical decomposition of the elements of 3D rendering into four major class categories: *Shapes* define geometric objects; *attributes* specify quality and visual appearance; *controllers* describe rendering techniques and rendering processes, and *rendering engines* evaluate shapes, attributes, and controllers. We have implemented our concepts in VRS, the *Virtual Rendering System*, as a portable C++ toolkit. VRS currently supports rendering packages such as OpenGL, PEX, XGL, Radiance, POV Ray, and RenderMan.

1 Motivation

3D rendering packages are of increasing importance for the development of applications in CAD, CAE, medical and scientific visualization. Both the quality and the speed of rendering techniques improved dramatically in the last few years. However, 3D software is still very immature. Developers of 3D applications are confronted with the following problems:

- Implementing applications on top of existing 3D rendering libraries (e.g. OpenGL, QuickDraw 3D, Direct 3D) is difficult because the application developer has to learn and understand a multitude of low-level data structures and subroutines provided by these packages. In particular, the programming interfaces of existing graphics libraries are oriented towards the rendering pipeline instead towards the developer's perspective, and they provide only a low level of abstraction. Even for experienced programmers it is difficult to work efficiently with these libraries.
- Rendering packages are not fine-grained object-oriented, most packages are not object-oriented at all. Object-orientation proved to be useful for computer graphics [22]; however, applications can only take full advantage of object-oriented design if the application programming interface (API) of the rendering library is also designed in an object-oriented fashion.
- Graphics applications cannot be easily adapated to a different rendering technique. To port applications, they must be redesigned completely. For instance, applications based on an immediate-mode library cannot be ported easily to radiosity-based packages.

Rendering libraries embody a huge amount of work and know how: they contain lots of well designed algorithms and support specialized graphics acceleration hardware which in general is hard to program. It would be very expensive to write one new object-oriented rendering system from scratch which supports the whole bandwidth of graphics hardware and graphics algorithms as the existing graphics packages do. This has been the motivation for us to seek for an object-oriented framework which wraps the functionality and generalizes the API of most of today's rendering packages.
VRS, the *Virtual Rendering System*, provides an extensible class hierarchy which defines 3D shapes, (visual) attributes, image synthesis controllers and rendering engines. Shapes, attributes and controllers are managed by rendering engines which provide a homogeneous interface for rendering commands. VRS offers easy access to complex capabilities of (non object-oriented) rendering systems. In particular, VRS facilitates low-level 3D programming with OpenGL because it provides higher-level classes and have no impact on the overall performance of an OpenGL-based application. VRS allows us to develop applications for high-quality rendering and real-time rendering within one framework without loosing control over the specific features of individual rendering packages due to its generic rendering commands.
We have implemented VRS as a portable C++ toolkit which currently supports OpenGL [14], XGL [19], the physically-based lighting and simulation system Radiance [21], POVRay [16], and Pixar's RenderMan [20]. Fig. 1 shows two snap shots (rendered with OpenGL and POVRay) of an animation of Coxeter's polyhedron {4,6|3} with hidden symmetries; the edges of the polyhedron are projected onto the canvas.

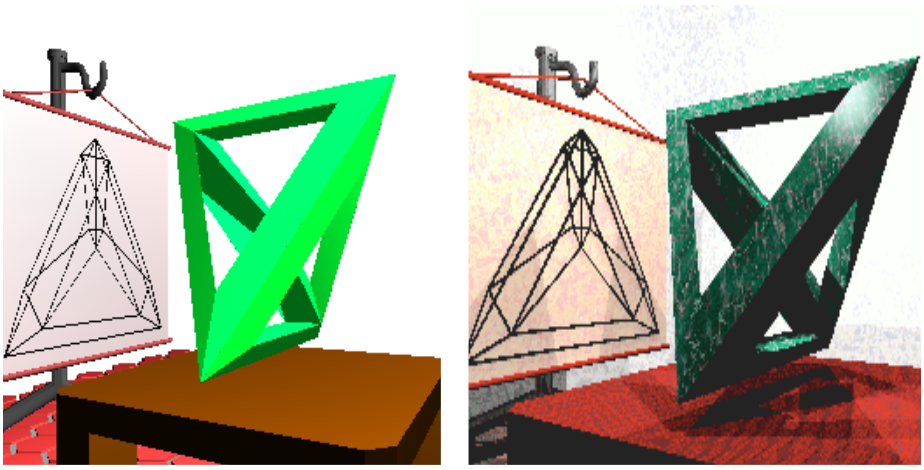

Fig. 1 Visualization of Coxeters {4,6|3}. Rendered with OpenGL and POVRay.

2 Concepts for a 3D Rendering Meta System

A *3D rendering meta system* is an abstract rendering system whose instances are actual rendering systems. It does not implement a specific rendering technique, its instances are implemented based on external 3D rendering packages. A rendering meta system defines a framework in which 3D rendering systems can be integrated and accessed in a generic fashion.

2.1 Design Criteria

Object-orientation is quite a natural vehicle for computer graphics. The object-oriented design of a rendering meta system has to satisfy different goals, in particular:

- *Encapsulate rendering commands by objects.* The features of 3D rendering packages should be available through objects. Graphics primitives (e.g. triangle set, NURBS) and graphics attributes (e.g. material) should be directly mapped to an appropriate class hierarchy. Even rendering features (e.g. accumulation buffer) or rendering techniques (e.g. shadow volume generation), should be encapsulated in objects.
- *Provide a general graphics class hierarchy.* The graphics classes should be as general as possible, i.e., they should be suitable for different rendering packages. Most 3D rendering packages provide common graphics primitives and graphics attributes. However, they differ with respect to high-level features (e.g., programmable shaders in RenderMan). A common class hierarchy allows us to exchange rendering techniques without the need to recode an application.
- *Provide an extensible graphics class hierarchy.* The class hierarchy should allow the developer to integrate rendering package specific features, application-specific graphics types, and new rendering packages. A rendering system has to provide built-in graphics classes, but this collection can never be complete, because new 3D applications may require new graphics types. Therefore, the meta rendering system should allow the developer to build new graphics classes which have the same “rights” as the built-in graphics classes and whose instances are possibly mapped to built-in graphics objects.
- *Maintain performance.* Compared to the direct use of a rendering package, the meta system should not introduce a significant performance overhead. In particular, real-time packages such as OpenGL should be integrated so that shape, attributes, and controllers can be mapped directly to library calls.

2.2 The Design of the Virtual Rendering System

VRS is a framework which specifies graphics classes and an abstract rendering engine. Instances of graphics classes are evaluated by a concrete rendering engine which performs an appropriate mapping of graphics objects to commands of the underlying 3D rendering package. The rendering meta system VRS can be viewed as a thin object-oriented layer on top of existing 3D rendering packages.

VRS Base Classes

The VRS class hierarchy is based on a logical decomposition of the elements of 3D rendering into four major classes (Fig. 2): *Shapes* define geometric objects; *attributes* specify quality and visual appearance of shapes and environments; *controllers* describe rendering features and techniques; and *rendering engines* manage and associate shapes, attributes, and controllers.

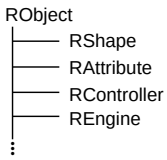

Fig. 2 VRS Base Classes.

Fig. 3 illustrates the relationship between VRS and OpenGL.

Flyweight Design

VRS classes can be characterized by their flyweight design [4] which ensures that they are as minimal and as small as possible, and that they can be used in large numbers and implemented efficiently [13]. For example, shape objects do not include any context information and do not make any assumptions about their geometric representation, e.g, a sphere stores its radius and center, but does not include a transformation matrix, a color, a rendering context or a polygonal surface approximation.

Full Encapsulation by a Functional Interface

Graphics objects should be accessed by pure functional interfaces in order to be able to access them through different application programming interfaces. For example, a graphics object might be used by a C++ API, accessed through a CORBA interface, or exported to an interpretative language like Tcl [15].

Export of Variables by Parameter Objects

Graphics objects which possess a time-dependent nature can be controlled by time-dependent constraints. If constraints are modeled by constraint objects, we have to access the variables of graphics objects in an object-oriented

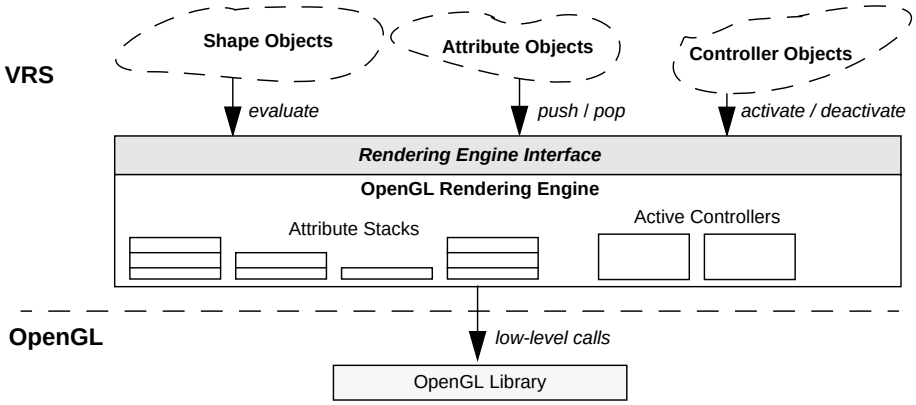

Fig. 3 VRS Objects and the OpenGL Rendering Engine.

way. To build generic constraint classes, constraint objects have to be based on (mostly elementary) parameter types (e.g., vectors, colors, floats, ...). Therefore, graphics objects must provide a way to export their parameters as *parameter objects*.

3 Rendering Engines

A rendering engine is an abstract device that works like a state machine. It provides a generic management for shapes, attributes, and controllers. The rendering engine class defines a uniform interface which consists of commands for activating and deactivating controllers, pushing and popping attributes, and evaluating shapes (Fig. 4).

```
class REngine : public RObject {
public:
 void activate (RController*, int position=0);
 void deactivate (RController*);

 void push (RAttribute*);
 void pop ();

 void evaluate (RShape*);
 ...
};
```

Fig. 4 Renderer Engine Interface.

3.1 Attribute Management in Rendering Engines

Rendering engines store and manage attributes by *attribute stacks*. Attribute objects can be pushed on and popped from stacks. A rendering engine maintains a separate attribute stack for each attribute class. An attribute object is referenced by the rendering engine when it is pushed, and dereferenced when it is popped. Since attributes are shareable objects, the attribute management is time- and space-efficient because no data have to be copied. Rendering engines provide a *generic attribute management*, i.e., if an attribute is pushed for which no attribute stack exists, a new attribute stack is automatically created. Therefore, applications can use the generic attribute management to handle their own attribute classes.
The rendering engine offers access to all current top attributes. A rendering algorithm decides which of these attributes to use. The direct *association of shapes and attributes* is left to the application in order to make different association concepts possible. For example, the application can provide graphics objects containing a list of attributes. Alternatively, we can organize shapes and attributes in directed acyclic graphs (or trees) and use a graph traversal to determine which attributes to apply to shapes. Both approaches can be implemented with the attribute management provided by rendering engines. For a discussion of hierarchical graphical scenes see Beier [1].

3.2 Shape Evaluation in Rendering Engines

A shape painter is an attribute of a shape class and encapsulates the rendering algorithm used to map the shape to the low-level rendering package.
The evaluation method of a rendering engine inquires the top shape painter attribute (there is one shape painter stack for each shape class). Then painter, shape, and rendering engine are passed as parameters to all active controllers. For example, a frame controller may call the rendering algorithm in the shape painter for the given shape. A picking controller, however, might check for intersection by directly calling the intersection method of the shape object. Finally, the rendering algorithm may inquire relevant attributes and the corresponding attribute painters in order to set up the attribute state in the rendering context. The attribute painters specify how an attribute is mapped to the low-level 3D rendering package. Alternatively, the rendering algorithm may decompose the shape into lower-level primitives, and invoke the evaluation method recursively.

A Minimal Example

This example (Fig. 5) uses the [incr Tcl] [12] API and shows how to compose images with VRS. Procedure *init* instantiates the graphics objects used in our scene; the objects are named by labels. The frame object is directly associated with the projection and orientation matrix. Procedure *redraw* implements the rendering commands to synthesize a texture cube viewed from the point (2,2,2) looking at the center. The rendering engine is passed as argument *e*. *init* and *redraw* are generic, i.e. they can be used for any type of rendering engine.

```
proc init {} {
 RColor gray {0.5 0.5 0.5}
 RTexture label "hellovrs.tif"
 RBox cube {-1 -1 -1} { 1 1 1 }
 RPerspective proj 45.0 0.1 100
 RLookAt orient {2 2 2} { 0 0 0 }
 RFrame frame proj orient
}


#
# e : Rendering Engine
#
proc redraw { e } {
 e activate frame
 e push gray
 e push label
 e evaluate cube
 e pop
 e pop
 e deactivate frame
}
```

Fig. 5 A Minimal VRS Example.

4 Controller Classes

Controller classes (Fig. 6) implement image synthesis techniques. A controller decides how to evaluate shapes when they are sent to the rendering engine. A rendering engine maintains a sorted sequence of controllers. Controllers depend on the 3D rendering packages used because these packages differ in the degree to which the image synthesis process can be controlled. For example, OpenGL and RenderMan provide rendering pipelines which can be programmed to a high degree whereas Radiance takes a scene description and synthesizes the image inside a black box.

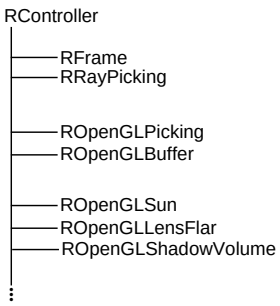

Fig. 6 Controller Class Hierarchy.

At least the following two controllers can be used for each rendering package: The *frame controller* is responsible for producing an image of a 3D scene, and the *ray picking controller* is responsible for intersecting the objects of a 3D scene with a ray. Specialized OpenGL controllers are responsible for the accumulation buffer, depth buffer, color buffer, and stencil buffer.

In addition, controllers can be used to encapsulate rendering algorithms such as multi-pass rendering or shadow volume generation. For example, a sun controller can model a sun-like light emission in the background image, and a lens flare controller can add the corresponding lens flares to the image (Fig. 7). Toolglasses (“magic lenses”) [3] can be implemented with OpenGL by controllers, too. An active toolglass controller restricts rendering to a region of the image plane, e.g. masked out by the stencil buffer.

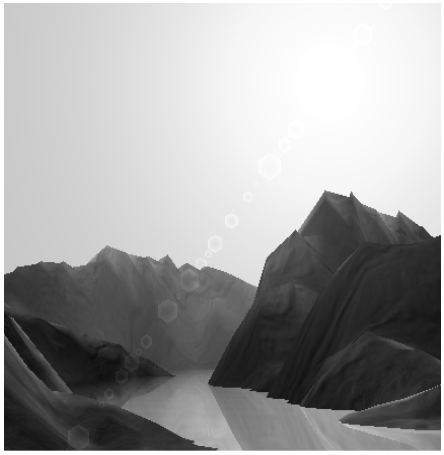

Fig. 7 Lens flare and sun controller applied to a terrain model using OpenGL.

Controllers have been introduced to provide a powerful extensibility to the VRS rendering pipeline and to give the developer full access to the capabilities of a 3D rendering package *through a homogeneous interface*. Furthermore, controllers are well-suited to wrap OpenGL rendering gems in reusable building blocks. These object-oriented wrapped visualization techniques are the main application of controllers.

5 Shape Class Design

In object-oriented rendering systems the design of the class hierarchy for shapes is crucial since it determines both the efficiency and reusability of the system (from a programmer's point of view). The utilization of shape classes is often restricted due to the following pitfalls.

Implementation-dependent Inheritance
Shape classes are derived by inheritance to share code although the derived classes are not true subclasses of their parent classes. If the correspondence in the internal representation is used as a criterion for inheritance, we get semantic inconsistencies which must be resolved by redefining operations of parent classes; i.e., operations of the parent classes must be overwritten in the derived classes in order to prevent that instances of the derived classes call these operations of the parent classes. For example, if we derive a cube class from a box class, we could call an operation of the box class for a cube object. However, the operations of the box class for setting the width, the height, and the depth contradict the cube semantics.

Renderer-dependent Inheritance
Shape classes are derived by inheritance because instances of the derived classes have the same (renderer-dependent) geometric representation as instances of the base class. For example, a rendering system could derive a sphere class from a triangle mesh class, and represent each sphere internally by a triangle mesh. However, there are rendering toolkits which do not use triangle meshes for spheres, instead they represent spheres in their analytic form. Using a common geometric representation as a criterion for subclassing leads to redundancy in the object representation and affects efficiency.

Non-Extensible Shape Classes
When application-specific shape classes cannot be added to the rendering system, i.e., if the set of built-in shapes is fixed, the application is forced to convert its data into built-in shapes. Consider for example an application managing 3D arrows. An arrow object is an application object. However, it has to be represented in the graphics layer in terms of built-in shapes. To represent an arrow, we could combine a cone shape and a cylinder shape. Two problems arise: (1) The application stores redundant information, i.e., for each arrow a cone shape and a cylinder shape. (2) The arrow semantics is lost in the rendering system. The semantics, however, could be useful to optimize rendering and intersection algorithms.

5.1 Built-in Shape Classes in VRS

VRS specifies a set of general purpose shape classes (Fig. 8) which cover shape types found in most 3D rendering toolkits. These built-in shape classes are guaranteed by VRS for all currently supported 3D packages. Shapes which cannot be converted into a primitive of the underlying 3D rendering package are automatically broken down into lower-level primitives.

We distinguish three main categories of shape classes.

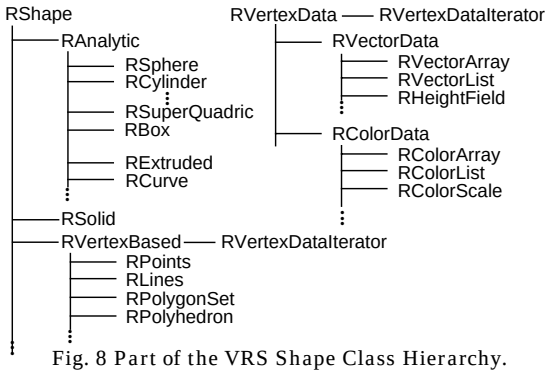

Fig. 8 Part of the VRS Shape Class Hierarchy.

- *Vertex-based shapes* are mainly defined by a set of characteristic vertices, e.g., point sets and polygons. Besides the coordinates, vertex data may include colors, texture coordinates, normals etc. Vertex data are provided by *vertex data iterators*. A vertex data iterator is an object which iterates over a collection of vertex data. The vertex data need not be stored in the shape object itself. This allows VRS to import vertex data directly from application data structures which only have to provide a suitable iterator object, or to use a functional specification of vertex data (e.g., height field and color scale).
- *Analytic shapes* are defined implicitly by a set of characteristic parameters. They encompass quadrics and extruded objects. Their specification includes the tessellation in both parameter dimensions, and they are sensitive to level-of-detail attributes.
- *Solids* represent closed bounded volumes, they can be combined by Boolean set operations, such as union, intersection, and difference. VRS uses a solid modeling library based on half-edge data structures [11].

5.2 Integrating New Shape Classes Based on Shape Painters

The rendering commands used to map shape objects to a low-level 3D rendering package are encapsulated in *shape painter classes*. For each shape class there is at least one corresponding shape painter class. Shape painters are attributes of shapes, they are bound at run time and form part of the state of a rendering engine.

The painter concept has the following consequences:

- VRS facilitates adapting new 3D rendering packages: most high-level shape types can be used immediately once the elementary shape painter classes are defined for the new rendering package. For high-level shape classes (e.g. BSpline, quadrics) we provide generic shape painter classes which transform a shape object into a set of lower-level shape objects.
- The visualization of shapes is separated from their data. Different shape painters may be provided for one shape class. For example, a cell array may be viewed as a grid or as a set of transparent cubes.
- Application-specific data can be visualized by new shape classes. Data have not to be converted into built-in shapes.
- For time-critical rendering, optimized shape painters which take advantage of the a priori known nature of the data can be specified for application-specific shapes.

6 Attribute Class Design

By attributes we mean all parameters and options which define visual or geometric properties of shapes and environments, e.g. textures, atmospheric effects, transformations etc.

6.1 Generic Attribute Management

Attributes depend to a high degree on the rendering technique actually used. Therefore, it is not possible to achieve a set of uniform attribute classes for a 3D rendering meta system. Instead, we provide a generic attribute management with the following goals:

- Represent attributes in an object-oriented fashion and decouple attribute management from shape management. Modeling attributes as objects in a separate class hierarchy supports extendibility because it allows applications to define specialized attribute classes. Representing attributes as objects permits to use object-oriented features for attributes. VRS models attributes as first-class objects. Like shape objects, they are graphics objects which can be referenced multiple times by other graphics objects.
- Provide a core set of attribute classes which are general enough to produce reasonable results for different rendering techniques. The degree to which an attribute can be evaluated by a specific rendering engine may, of course, differ according to the available rendering capabilities. In general, all rendering packages based on a local illumination model provide similar attributes due to the same shading techniques. Rendering packages based on a global illumination model provide a more complex set of attributes with respect to reflection, transmission, and emission of light.
- Provide a generic attribute management in order to handle new renderer-specific and application-specific attribute classes. For example, VRS provides Radiance and RenderMan attribute classes to make the specific shading and lighting features of these rendering packages available. Application-specific attributes might be evaluated by application-specific shape painters.

6.2 VRS Attribute Class Hierarchy

VRS defines six main groups of attributes – surface shaders, volume shaders, light sources, transformations, shape painters, and attribute painters (Fig. 9). *Surface shaders* define the surface characteristic and surface shading (e.g., matte, metal or plastic). *Volume shaders* specify attributes applied to the environment, e.g. depth cueing and fog. *Light sources* specify the illumination of 3D scenes. *Transformations* represent 4×4 matrices and are used to modify the modeling coordinate system. *Painters* encapsulate rendering algorithms used to map shapes and attributes to 3D rendering packages. Shape painters are attributes of shapes, attribute painters are attributes of non-painter attribute classes.

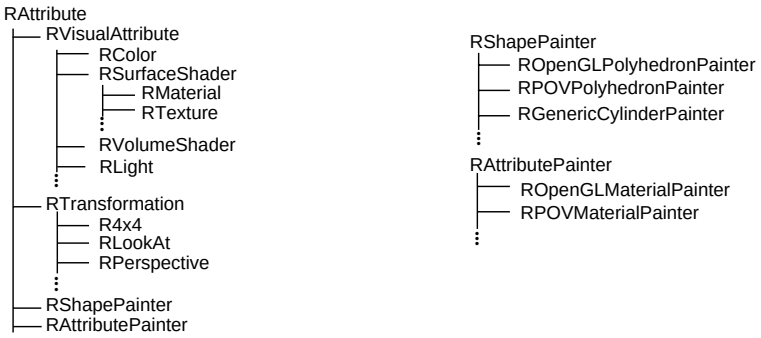

Fig. 9 Part of the VRS Attribute Class Hierarchy.

7 VRS Implementation

VRS has been implemented as a C++ library for NT and Unix. It can be used with the C++ API and an optional [incr Tcl] [12] interface which provides developers interactive access to VRS. [incr Tcl] is an object-oriented extension of Tcl/Tk [15] and proved to be a valuable tool for the interactive testing of the toolkit and rapid prototyping.

Most rendering packages are adapted using the generic shape painter classes. Specific rendering features are represented by specialized attribute classes and/or attribute painter classes. We provide optimized shape and attribute painters for OpenGL; the overall performance of a graphics application compared to a direct implementation with OpenGL is not affected significantly: an average overhead of approximately 5% results from 2 - 4 virtual function calls per shape necessary to invoke the rendering algorithm. The full functionality of OpenGL can be accessed via the built-in classes.

8 Related Work

A different approach to an extendible architecture for rendering is taken by systems such as Spectrum [9] and Vision [17]. They support multiple rendering techniques based on an object-oriented model of the physical rendering process. The VRS approach starts from the opposite “direction”: it designs an extensible virtual rendering engine whose object model allows us to *use different external rendering toolkits*, but *does not implement* such a rendering toolkit. Therefore, VRS is called a *meta system*. This pragmatic approach allows VRS to guarantee ease of use, exchangability and extensibility of rendering systems without the need to rewrite application code. It would be very interesting to see how a direct and native implementation of an object-oriented rendering system could profit from the VRS object-model. Object-oriented graphics systems, e.g., GRASSY [6], GROOP [10], OpenInventor [18], TBAG [7] and Generic [2] do not concentrate on designing a general API for 3D rendering but intend to be high-level graphics systems. VRS, on the other side, could support the implementation of a graphics system, e.g., MAM [5] has been developed based on VRS.

9 Conclusions and Future Work

VRS implements the concepts of a 3D rendering meta system described in this paper. VRS allows us to adapt 3D applications efficiently and quickly to different rendering packages and therefore to satisfy different visualization needs in industrial and scientific visualization applications. Our toolkit offers an efficient encapsulation of the industry standard OpenGL in an object-oriented fashion. VRS is currently used as a software platform in several visualization projects in medicine, chemistry, physics and the geo-sciences. Due to the simplicity and transparency of its API, it can be used for educational purposes as well. A native Java version of VRS is under development.

Acknowledgements. The VRS project is supported by the Ministry of Science and Research, Northrhine-Westphalia, Germany. We thank Peter Serocka (VisLab University Bielefeld) for Fig. 7.
WWW-Site: http://wwwmath.uni-muenster.de/~mam.

References

[1] E. Beier, *Issues on Hierarchical Graphical Scenes*. In: R. Veltkamp and E. Blake, eds., *New Directions in Computer Graphics*. Springer-Verlag, 1995.

[2] E. Beier, U. Bozzetti. *A Generic Graphics Kernel and a Customized Derivative*. Available from ftp://metallica.prakinf.tu-ilmenau.de/pub/PROJECTS/GENERIC/dublin.ps.gz.

[3] E. Bier, M. Stone, K. Pier, W. Buxton, T. DeRose. *Toolglasses and magic lenses: the see-through interface*. Proceedings of SIGGRAPH'93, pp. 73-80.

[4] P. R. Calder, M. A. Linton, *Glyphs: Flyweight Objects for User Interfaces*. Proceedings of the ACM SIGGRAPH Third Annual Symposium on User Interface Software and Technology, 1990.

[5] J. Döllner, K. Hinrichs, *Object-Oriented 3D Modeling, Animation and Interaction*. The Journal of Visualization and Computer Animation, Vol. 8, pp. 33-64, 1997.

[6] P. K. Egbert, W. J. Kubitz, *Application Graphics Modeling Support Through Object -Orientation*, Computer, October 1992, pp. 84-91.

[7] C. Elliot, G. Schechter, R. Yeung, S. Abi-Ezzi SunSoft, Inc. *TBAG: A High Level Framework for Interactive, Animated 3D Graphics Applications*. Proceedings of SIGGRAPH '94, pp. 421-434.

[8] D. W. Fellner, *Extensible Image Synthesis*. In: 4th EuroGraphics Workshop on Object-Oriented Graphics, 1994, pp. 1-18.

[9] A. Glassner, *Spectrum: a Proposed Image Synthesis Architecture*. SIGGRAPH '91 Frontiers in Rendering course notes, Jul. 1991.

[10] L. Koved, W. L. Wooten, *GROOP: An object-oriented toolkit for animated 3D graphics*. ACM SIGPLAN NOTICES OOPSLA'93, Vol. 28, No. 10, October 1993, pp. 309-325.

[11] M. Mäntylä. *An Introduction to Solid Modeling*. Computer Science Press, 1988.

[12] M. J. McLennan, *[incr Tcl]: Object-Oriented Programming in Tcl*, Proceedings of the Tcl/Tk Workshop 1993, University of California, http://www.wn.com/biz/itcl.

[13] M. A. Linton, J. M. Vlissides, and P. R. Calder. *Composing user interfaces with InterViews*. *IEEE Computer*, pp. 8-22, February 1989.

[14] J. Neider, T. Davis, M. Woo. *OpenGL Programming Guide*. Addision-Wesley, 1993.

[15] J. K. Ousterhout, *Tcl and the Tk Toolkit*, Addison-Wesley, 1994.

[16] POV Team. *Persistency of Vision Ray Tracer (POV-Ray)*. Version 1.0, Technical Report, 1991.

[17] P. Slusallek. *Vision - An Architecture for Physically-Based Rendering*. PhD thesis, University Erlangen-Nuernberg, April 1995.

[18] P. S. Strauss, R. Carey, *An object-oriented 3D graphics toolkit*. SIGGRAPH'92 Proceedings, Vol. 26, No. 2, pp. 341-349.

[19] Sun Microsystems. *The Solaris XGL Graphics Library*. Sun Microsystems Inc., Mountain View, CA, Februar 1993.

[20] S. Upstill. *The RenderMan Companion. A Programmer's Guide to Realistic Computer Graphics*. Addison-Wesley, 1990.

[21] G. J. Ward, *The RADIANCE Lighting Simulation and Rendering System*, Proceedings of SIGGRAPH' 94, pp. 459-472.

[22] P. Wisskirchen. *Object-Oriented Graphics: From GKS and PHIGS to Object-Oriented Systems*. Springer-Verlag, Berlin, 1990.